On separation pairs and split components of biconnected graphs

Sven Mallach*

Institut für Informatik
Universität zu Köln, 50969 Köln, Germany

21th June 2011.

Abstract

The decomposition of a biconnected graph G into its *triconnected components* is fundamental in graph theory and has a wide range of applications. Based on a palm tree of G, the algorithm by Hopcroft and Tarjan [7] is able to compute them in linear time if the corrections of [6] are applied. Today, the algorithm is still considered very hard to understand and proofs of its correctness are technical and challenging.

The article at hand provides a more comprehensive description of the algorithm, making it easier to understand and implement. Its correctness is validated by explicitly mapping the algorithmic detection criteria to the graph-theoretic characterization of *type-1* and *type-2 separation pairs*. Further, it reveals further errors and inaccuracies in the common definitions. This includes the description and proofs of further properties and relationships of separation pairs. The presented results also answer the question whether and under which preconditions type-1 and type-2 pairs can be computed separately from each other.

1 Introduction

Determining vertex connectivity is a fundamental graph theoretical problem with applications in many domains. Besides immediate applications for determining the structure and degree of connectivity in networks, triconnectivity is of special interest in graph drawing, since planar triconnected graphs have a unique embedding in the sphere. Many drawing algorithms (e.g., [2, 8]) make use of the decomposition of a biconnected graph into its triconnected components (usually represented as SPQR-tree [5, 1, 6]), which allows to represent all planar embeddings of the graph.

The major task to compute the triconnected components of a graph $G = (V, E)$ is to determine its *separation pairs*. Two vertices $a, b \in V$ are called a separation pair if and only if $G - \{a, b\}$ is not connected and there are neither exactly two components resulting one of which consists of a single edge

*Please address correspondence to mallach@informatik.uni-koeln.de

nor exactly three components resulting each of which consists of a single edge [7]. Though based on depth-first search (DFS), the first linear-time algorithm given by Hopcroft and Tarjan [7] in 1972, is considered to be sophisticated to retrace and implement, but is still state-of-the-art. Several errors have not been discovered and corrected until the work by Gutwenger and Mutzel in 2000 [6]; see also [5] for a more in-depth description.

Hopcroft and Tarjan distinguish *type-1* and *type-2* separation pairs and characterize them in terms of graph theory (using cycles and segments) and by algorithmic tests. In [7], a proof is provided that pairs of vertices which pass the algorithmic tests also conform to the graph-theoretic definition of separation pairs. However, especially for the type-2 case, this is challenging to see. Recently and independently from this work, a first effort towards a more comprehensive explanation has been achieved in a master's thesis [9].

The aim of this article is to offer an accessible description by explicitly mapping the algorithmic to the graph-theoretic criteria and to unveal further inaccuracies in the common definitions used in relation to the identification of triconnected components. Therefore, we describe special cases of type-1 pairs that are correctly determined by the algorithm, but not covered by their structural definition. Then we prove properties of type-2 pairs that have not been part of any previous algorithmic characterization, but which are necessary in order to be correct. From the algorithmic point of view, we also answer the question whether type-1 and type-2 pairs can be computed separately from each other and under which preconditions this is possible.

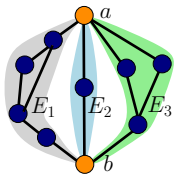
As a further contribution, we describe a relationship which is, to the best of our knowledge, previously unknown: For every separation pair $\{a, b\}$ that is detected as being of type-2, we can perform a different DFS traversal in which $\{a, b\}$ will be detected as being of type-1. Theoretically, this allows for an algorithm performing exclusively type-1 splits. This would be promising, as finding type-2 pairs requires additional data structures, sorting, comparisons and even full passes over the vertex and edge sets. Hence, if there was a way to construct a type-2-free DFS tree for every graph a new simple and efficient algorithm would be conceivable. However, we give a counter-example showing that there exist graphs for which no such tree exists.

Related work Besides the already mentioned literature, some alternative approaches to compute the triconnected components were considered in research. In the 1980s, Vo [14, 13] and Williamson [15] presented two slightly different classifications of separation pairs, one using *segment graphs* and one using *bridge graphs*, respectively. Triconnectivity also received some attention within the PRAM community, e.g. in [10, 4], but the proposed algorithms turned out to be even more complicated and barely applicable in practice. Similarly, computing separation pairs via transformations from 3-edge-connectivity, like in, e.g., [11], neither yields more simplicity nor ease of implementation.

The sequel of this paper is organized as follows: In Sect. 2, we reproduce the basic concepts and notations to handle the detection of separation pairs using DFS. Sections 3 and 4 present the main results. Finally, we conclude in Sect. 5.

2 Preliminaries

2.1 Separation pairs and split operations in biconnected graphs

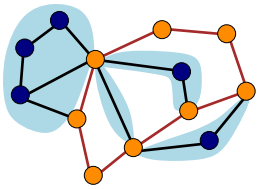


Let $G = (V, E)$ be a biconnected multigraph, and $a, b \in V$. We may partition E into *separation classes* $E_1, E_2, \ldots, E_k$, so that two edges are in the same class if they lie on a common path which does not comprise a or b except as an end vertex. Suppose there are at least two separation classes. Then $\{a, b\}$ is a *separation pair*, if there are neither exactly two classes one of which consists of a single edge nor exactly three classes each of which consists of a single edge [7].

Let E' be the union of a subset of the E_i and $E'' = E \setminus E'$, such that $|E'| \geq 2$ and also $|E''| \geq 2$. A *split operation* replaces G by two *split graphs* $G_1 = (V(E'), E' \cup \{a, b\})$ and $G_2 = (V(E''), E'' \cup \{a, b\})$ where (a, b) is a new *virtual edge*. If, in addition, E' or E'' consists of a *single* split class (i.e, is minimal w.r.t. split operations) then the split is called a *Tutte split* [12].

The algorithm of Hopcroft and Tarjan exclusively performs Tutte splits. The recursive application of split operations yields the *split components* of G which are either a pair of vertices with three edges (bonds), triangles or triconnected subgraphs. Bonds and triangles are valid split components due to the above definition of separation pairs which mainly serve to decompose a bi- but not triconnected graph in a well-defined manner. It prohibits split graphs consisting of one or two edges (a, b) only, but explicitly permits split graphs with two or three vertices and at least three edges. The split components of G are not unique. Merging bonds and triangles with a common virtual edge into maximal bonds and polygons, one yields the unique *triconnected components* of G [7]. These do not necessarily apply to common definitions of 3-vertex-connectedness, such as Menger's theorem [3]. A bond is a valid triconnected component though it has less than three vertices. Even more, a quadrangle is a valid triconnected component even though removing two non-adjacent vertices 'disconnects' it.

2.2 Cycles and segments of biconnected graphs



The linear-time triconnectivity algorithms discussed here mainly rely on the analysis of cycles in a simple biconnected graph $G = (V, E)$ while split components arising from multi edges are processed seperately in advance. If we find a cycle C in G and conceptually remove it, G decomposes into connected components which are called *segments* relative to C. More precisely, segments are defined by the set of vertices incident to the edges of the component, i.e., the vertices of attachment on C are also considered to be part of the segments.

To consider cycles is efficient due to the following property (proved in [7]) which allows for an algorithm that successively builds new cycles from a previous one and one of its segments using DFS.

Theorem 1. *Let $\{a, b\}$ be a separation pair and C a cycle in G, then either a and b lie both on C or both in the same segment relative to C.*

2.3 Depth-first search and palm trees

Traversing a graph with DFS means decomposing it into *simple paths*, consisting of a sequence of tree arcs followed by one back arc. We use the following notation: If (u, v) is a tree arc denote it with $u \to v$, if it is a back arc (subsequently called a *frond*) with $u \hookrightarrow v$. Similarly, denote a path from u to v consisting of zero, one or multiple tree arcs with $u \to^* v$.

The tree constructed by DFS is a so-called *palm tree* $P = (V, A)$ [7] which is a directed multigraph where each arc $(u, v) \in A$ is either a tree arc or a frond such that the tree arcs form a spanning tree and if $u \hookrightarrow v$, then also $v \to^* u$. In P, every vertex v has a set of descendants $D(v)$, a degree $deg(v)$, a unique DFS number $num(v)$ and a unique predecessor $parent(v)$. The lowpoint of a v corresponds to the lowest and second lowest numbered vertices that are reachable in P from v by simple paths, i.e., $lowpt_1(v) = \min\{num(v)\} \cup \{num(w) | v \to^* \hookrightarrow w\}$ and $lowpt_2(v) = \min\{num(v)\} \cup \{num(w) | v \to^* \hookrightarrow w\} \setminus \{lowpt_1(v)\}$ for $v \in V$.

The ordinary DFS arbitrarily chooses one unseen descendant to process next. Instead, due to Theorem 1, we would like to have an initial cycle and each following simple path to end at the vertex with the lowest possible DFS number and to share at most its start and end vertex with previous paths. Then each such path $x \to^* y \hookrightarrow z$ defines a segment S relative to its parent cycle C and can itself be extended to a cycle by adding the tree-arc path $z \to^* x$. In the notation of [14, 13, 15], we call $z \to^* x$ the *span* of S, $span(S)$ and $span(S) \setminus \{z, x\}$ the open span of S, $ospan(S)$. The vertices z and x belonging to C and S are also referred to as $low(S)$ and $tail(S)$, respectively.

In order to gain the desired order of cycles we need to traverse T twice, orienting the second traversal at the *lowpt* information calculated in the first pass. Therefore, between the passes, the adjacency lists $A(v)$ of all vertices v are sorted by $lowpt_1(w)$, if $v \to w$, or $num(w)$ if $v \hookrightarrow w$. If vertices $w_i, \ldots, w_k$ have the same $lowpt_1$-value $num(u)$ then there is some index $j \leq k$ such that $lowpt_2(w_i) < v$ for all $i \leq j$ and $lowpt_2(w_i) \geq v$ for all $i > j$. Fronds $v \hookrightarrow u$ will be placed in between all arcs to vertices $w_i, i \leq j$ and $w_i, i > j$. Later, this will ensure a correct determination of split components stemming from type-1 pairs $\{u, w_i\}$ with $i < j$ since all edges (v, w_i) with $lowpt_2(w_i) \geq v$ appear consecutively in $Adj(v)$. Sorting the adjacency lists like this can be done in linear time using bucket sort as described in [6]. Afterwards, all vertices except the root are renumbered according to the new adjacency structure. The children $w_1, \ldots, w_n$ of a vertex v in the order of $A(v)$ are assigned the numbers $num(w_i) = num(v) + |D(w_{i+1}) \cup \cdots \cup D(w_n)| + 1$.

Let $u_0, \ldots, u_k$ be a path. If for every $i, 1 \leq i \leq k$, u_i is the first vertex in the adjacency list of u_{i-1}, then u_k is called a *first descendant* of u_0. Due to the adjacency structure, a path of first descendants corresponds to the path that reaches back at furthest and will be traversed first.

3 Structure and algorithmic detection of separation pairs

In this section, the segment-based and algorithmic detection criteria for type-1 and type-2 separation pairs $\{a, b\}$ as given in [7] are revisited. First, the algorithmic tests are mapped to the segment-based criteria. For type-1 pairs,

we find special cases that are not covered by the segment-based characterization whereas, for type-2 pairs, we prove new necessary conditions that have to be satisfied algorithmically.

For all the stated conditions it is assumed that $num(a) < num(b)$ and that there is a palm tree with an adjacency structure and numbering as described in Sect. 2.3. Within the illustrating figures in this section solid arcs (u, v) refer to a direct adjacency, dotted arcs to paths $u \to^* v$ and unnamed segments represent parts of a palm tree that may or may not exist without influencing validity of the respective scenario. Further, a subdivided palm tree with replicated separation-pair vertices a and b shall represent a split operation while the virtual edges to be added are not visualized.

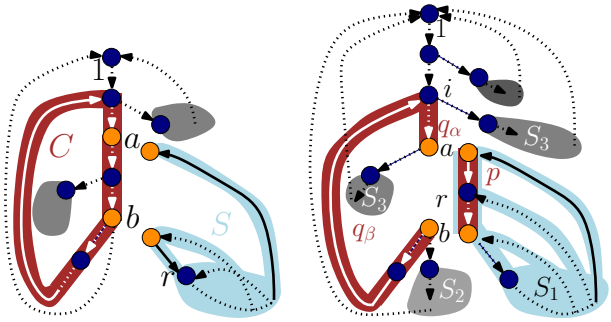


Figure 1: A typical type-1 pair (left) and a typical type-2 pair (right).

3.1 Type-1

3.1.1 Mapping the algorithmic to the segment-based criteria

For the segment-based characterization, we consider a cycle C and find a type-1 pair if a segment S relative to C consists of at least 2 edges, has only a and b in common with C and there exists another vertex outside S which is not equal to a or b [7]. An example is shown in the left of Fig. 1.

Definition 1 (Type-1 separation pair, algorithmic conditions)**.**

> There exist vertices $r \neq a, b$ and $s \neq a, b$ so that $b \to r$ and $s \notin D(r)$
> $lowpt_1(r) = num(a),\ lowpt_2(r) \geq num(b)$

Theorem 2. *If the algorithmic criteria for type-1 pairs are satisfied, the segment-based criteria are also satisfied.*

Proof. For type-1 pairs this is straightforward: Assuming a and b to lie on C, we enter a segment relative to C by traversing the arc $b \to r$. If $lowpt_1(r) = num(a)$ and $lowpt_2(r) \geq num(b)$ then a and b are the only vertices on C that the segment connects to, i.e. removing these vertices will split it. It is clear that $r \neq a, b$ and that there must exist another vertex $s \neq a, b$ because otherwise the second split graph would be empty. □

However, the segment-based requirements are even more restrictive than the algorithmic test, as they refer to a cycle C and a segment S relative to it, so that

$V(S) \cap V(C) = \{a, b\}$. We will now give small examples for two valid special cases of type-1 pairs where this is not the case.

3.1.2 Special cases not covered by the segment-based criteria

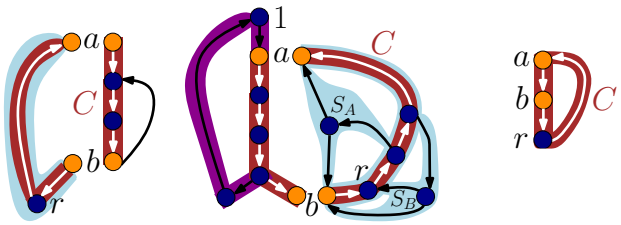


Figure 2: A type-1 pair with r lying *on* the cycle (left), with multiple segments having *more than two* vertices in common with the cycle (middle) and a special case satisfying the conditions but not admitting a type-1 pair (right).

At first, the vertex r satisfying the algorithmic criteria may itself lie on C. Consider the simple palm tree with root a in the left of Fig. 2. There is *no segment* that has two vertices in common with C. But r which is lying *on* C satisfies the conditions $lowpt_1(r) = num(a)$ and $lowpt_2(r) \geq num(b)$. The triangle $b - r - a$ created by the virtual edge $b \hookrightarrow a$ is a valid split component.

Secondly, the image in the middle of Fig. 2 shows two segments S_A and S_B that share *more than two vertices* with their cycle C. Following [14], they are said to be *directly linked*, i.e., $span(S_A)$ is not a subset of a path between two attachments of S_B on C and vice versa. Such segments must be drawn on different sides of C in any planar embedding [14]. Hence, for readability, the graph is not shown as a palm tree. In [13], Vo calls the illustrated case a *type-2a separation pair*. S_A and S_B belong to the same split component of $\{a, b\}$ which will be correctly detected to satisfy the type-1 conditions when backtracking from $b \to r$.

A third issue is a vertex pair $\{a, b\}$ satisfying the type-1 conditions while $parent(b) = a$ is the root of the palm tree and there is no unvisited tree arc left in $A(b)$. A minimal example is shown in the right of Fig. 2. Here, one split graph would consist of a and b connected by a tree arc and a virtual frond which is an invalid split operation according to the definition from Sect. 2.1. Although, we could not find this case differentiation in any previous segment-based definition, the corrected version of the algorithm of Hopcroft and Tarjan already checks for this special case [6].

3.2 Type-2

3.2.1 Mapping the algorithmic to the segment-based criteria

For the segment-based characterization, we again consider a cycle C with two vertices a and b. As a convention, we partition C into two halves: The tree-arc path $a \to^* b$ which is referred to as p, and the remaining part $b \to^* \hookrightarrow \to^* a$, called q. We can further divide q into the path from b to the frond and call it q_β, and the tree-arc path to a, q_α (cf. the right of Fig. 1). We call the segments connected to C by tree arcs just *the segments of* C.

Accordingly, $\{a, b\}$ is a type-2 pair if there is no segment of C that *simultaneously* contains a vertex on p and q, and both half-cycles consist of at least one more vertex [7]. Intuitively, a and b divide C such that every segment connected to one half has no connection to the other *without using* a *or* b. One can also say that removing $\{a, b\}$ isolates a vertex r (which is on p) from $parent(a)$ (which is on q). Before we establish the relationship between both characterizations, notice that the type-2 pairs have a special case, namely the parent a and descendant b of a vertex r with $deg(r) = 2$. For such a pair, the algorithmic conditions are trivially satisfied. We subsequently always refer to the non-trivial type-2 case.

Definition 2 (Type-2 separation pair, algorithmic conditions)**.**

$$
\begin{aligned}
&num(a) > 1 \text{ and there exists a vertex } r \neq a \text{ with } a \to r \to^* b \\
&b \text{ is a first descendant of } r \\
&\text{for all } x \hookrightarrow y \text{ with } num(r) \leq num(x) < num(b) \\
&\text{holds } num(y) \geq num(a) && (FC_1) \\
&\text{for all } x \hookrightarrow y \text{ with } num(a) < num(y) < num(b) \text{ and } b \to w \to^* x \\
&\text{holds } lowpt_1(w) \geq num(a) && (FC_2)
\end{aligned}
$$

Within the algorithm, the satisfaction of these criteria is mainly determined via the so-called *triple stack* (TSTACK). We will come back to TSTACK in Sect. 3.3.2, but for now, let us assume we have found a pair $\{a, b\}$ for which all conditions are satisfied.

Lemma 1. *If the algorithmic criteria for type-2 pairs are satisfied and* $\{a, b\}$ *is removed, then* p *is not connected to* q*.*

Proof. From the property that b is a first descendant of r we can conclude that on the path $r \to^* b$ there is no vertex x with $lowpt_1(x) < lowpt_1(b)$. Due to biconnectivity, for every vertex v except the root holds $lowpt_1(v) < num(v)$. Hence, $lowpt_1(b) \leq num(a)$, i.e., the first descendant property already forbids fronds like in condition (FC_1) for such vertices x. However, there could be a vertex x' with $lowpt_1(x') = lowpt_1(b)$ that still connects p to q_α. But due to the numbering, as b is the first descendant, a segment with such a vertex x' must have lower numbers than b has. Hence, (FC_1) is mandatory for x', i.e., there must not be a frond to a vertex y with $num(y) < num(a)$. It further follows from the structure of a palm tree that connections from p to q_β must proceed via a or b. As a consequence, all paths to q from p and its segments must proceed via a or b. □

Lemma 2. *If the algorithmic criteria for type-2 pairs are satisfied and* $\{a, b\}$ *is removed, then* q *is not connected to* p*.*

Proof. As $num(a) \neq 1$ there is at least one vertex v with $num(v) < num(a)$. Let I be the set of such vertices. Since (FC_1) holds and b is a first descendant of r, there is at least one path from b to a vertex in I due to biconnectivity. Notice that one of these paths is q_β. For all descendants w of b that are connected to I by a frond, (FC_2) ensures that there is no *direct* path (that is, without using a or b) to p at the same time. Hence, q_β is not connected to p. For q_α or segments of it, it follows directly from the structure of a palm tree that they are not simultaneously connected to p except by using vertex a or b. □

Theorem 3. *If the algorithmic criteria for type-2 pairs are satisfied, the segment-based criteria are also satisfied.*

Proof. The part dealing with absence of a segment that is simultaneously connected to a vertex on p and q follows directly from Lemmata 1 and 2. It remains to show, that they are also not empty which is immediate, as p at least comprises r and q at least comprises $parent(a)$.

□

As a side product we have described the only two possible kinds of segments starting in b if $\{a, b\}$ is assumed to be a type-2 pair (cf. the right of Fig. 1):

(S_1) Segments that are connected to p but not to some vertex i with $num(i) < num(a)$.

(S_2) Segments that are connected to some vertex i with $num(i) < num(a)$ but not to p.

Further, let i be the start and end vertex of the cycle. Suppose $num(i) \neq 1$ and no descendant of b has a frond to a vertex of the path $1 \rightarrow^* i$. Then, in order to maintain biconnectivity, there must be fronds to this path starting either from q_α, segments of it or from a subtree of a not containing r. We call these segments of type (S_3) (cf. the right of Fig. 1). Finally, we know that the split off component of $\{a, b\}$ consists of all edges on p, segments connected to p and all segments of type (S_1) (cf. Fig. 1 and [5]).

3.3 Implementation of the detection criteria for separation pairs

It has been just pointed out that the satisfaction of the algorithmic conditions for a type-2 pair also leads to the satisfaction of their segment-based definition. However, we will emphasize in Sect. 3.3.1 that the algorithmic properties as such define a superset of the vertex-pairs that can be correctly split as type-2 pairs. In other words, an additional condition has to be added to the algorithmic detection criteria in order to correctly specify the real type-2 pairs. In Sect. 3.3.2 we explain in detail how stacks are used in order to perform the separation pair detection in linear time.

3.3.1 The path starting property

Consider the palm tree in the left of Fig. 3. The pair $\{3, 7\}$ satisfies all conditions of a type-2 pair. But the algorithm in [7, 6] would not split it and instead discover $\{2, 5\}$ as a type-1 pair, like in the right image of Fig. 3. After that, a split of $\{3, 7\}$ is not possible anymore and if the new virtual edge $5 \hookrightarrow 2$ had existed before, it would have violated (FC_1) for $\{3, 7\}$. Hence, $\{3, 7\}$ does not correspond to a type-2 pair which means that the stated algorithmic criteria has to be enhanced by a further condition which will be pointed out now.

Lemma 3. *Let T be the palm tree of a simple biconnected graph and v with $num(v) \neq 1$ be a vertex in T. Then $deg(v) = 2$ if and only if neither a path starts at v nor a frond has v as its head.*

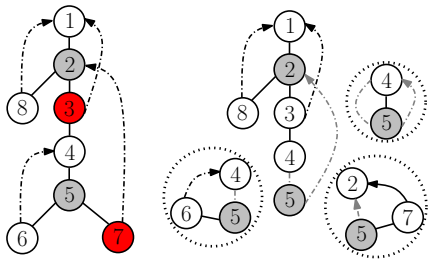


Figure 3: A vertex pair $\{3, 7\}$ satisfying the type-2 conditions beforehand (left) and type-1 splits leading to their subsequent violation (right).

Proof. From the definition of T each of its vertices, except the root, has exactly one parent. Suppose v is not a leaf. If it has exactly one outgoing arc it does not start a path. If v is also not head of a frond, then $deg(v) = 2$, otherwise $deg(v) \geq 3$. If v has multiple outgoing arcs, only one of them continues the path from $parent(v)$. Every other arc starts a path and $deg(v) \geq 3$. If v is a leaf, it cannot be head of a frond and it must have at least one frond due to biconnectivity. If it has only one frond, $deg(v) = 2$, otherwise it has multiple ones and therefore starts a path, so $deg(v) \geq 3$. □

Lemma 4. *Let $\{a, b\}$ a type-2 pair in T. Then $deg(b) > 2$ in T.*

Proof. Suppose $deg(b) = 2$. If b is a leaf in T, let v be the vertex with $num(v) = lowpt_1(b)$. Then $\{v, parent(b)\}$ is a type-1 pair. Otherwise, b is not a leaf in T. Then $\{parent(b), son(b)\}$ is a trivial type-2 pair. In both cases, b will be part of the corresponding triangle split component only. Therefore, b cannot contribute to any other separation pair as it will not further be part of the palm tree. □

Theorem 4. *Let T be a palm tree. If $\{a, b\}$ is a non-trivial type-2 pair in T, then b starts a path in T.*

Proof. Firstly, suppose b is a leaf of T. Following Lemmata 3 and 4, b has multiple outgoing fronds and therefore starts a path. Now suppose b is not a leaf of T and no path starts in b. If the only arc leaving b is a frond then b is a leaf $\lightning$. If it is a tree arc then $deg(b) = 2\lightning$. In any other case, b starts a path. □

So let us come back to the problematic case shown in Fig. 3. It can be generalized to the case where a type-1 pair $\{c, d\}$ splits a segment S and a pair $\{a, b\}$ satisfies the algorithmic type-2 pair criteria while a lies on the first half-cycle between c and d and $b \in S$ at the same time.

Adding the path-starting property to the algorithmic detection criteria for type-2 pairs solves this scenario. From Sect. 3.2.1 we know that if $\{a, b\}$ satisfies the conditions of a type-2 pair, a path starting in b can only define segments of the types (S_1) and (S_2). Let r be the successor of d in S with $lowpt_1(r) = c$ and $lowpt_2(r) \geq d$ and suppose $b \in S$. If the segment started in b is of type (S_1), then it violates the condition $lowpt_2(r) \geq d$. If it is of type (S_2), then it violates the condition $lowpt_1(r) = c$. This means that both cases where b starts a path lead to a contradiction to either the assumption that $b \in S$ or that $\{c, d\}$ is a type-1 pair. Further, due to Theorem 4, adding the path-starting property

does not incorrectly restrict the set of type-2 pairs, as such a path must always exist. Although it has not been stated as an algorithmic criterion before it is in fact already used by the algorithm of Hopcroft and Tarjan as we will see in Sect. 3.3.2.

3.3.2 The role of ESTACK and TSTACK

The algorithm by Hopcroft and Tarjan makes use of two stacks in order to achieve a linear-time detection of separation pairs and their corresponding split components.

The first one is the *edge stack* (ESTACK). Each edge (v, w) is pushed on top of ESTACK after recursion tracks back from vertex w, immediately before the separation pair checks take place (cf. [7, 6]). Hence, as an invariant, ESTACK contains all edges that have already been visited in the palm tree but not yet assigned to a split component. If a separation pair $\{a, b\}$ is found, not the entire set of edges on ESTACK is assigned to the corresponding component, but only the consecutive edges that have their endpoints within a certain range of DFS numbers. This range can be computed from $num(a)$, $num(b)$ and the number of descendants of the adjacent vertices of b that belong to the respective segments.

The second and much more involved stack is called *triple stack* (TSTACK). TSTACK is used to maintain the first descendant property and to check for the satisfaction of (FC_1) and (FC_2) while backtracking in the palm tree. It consists of triples $(h, num(a), num(b))$ where $\{a, b\}$ is a potential type-2 pair and h is the number of the highest numbered vertex in the component that would be split off, if $\{a, b\}$ is found to be a real type-2 pair. Recall the definition of type-2 pairs: There may be arbitrary segments S (of type (S_1)) connected to b that have $low(S) \geq num(a)$, i.e., would belong to the split component of $\{a, b\}$, so h is necessary to identify the last edge on ESTACK that belongs to these segments. To ease notation, we will identify vertices with their DFS number, i.e., write (h, a, b) instead of $(h, num(a), num(b))$.

Consider a cycle C. The algorithm will recursively traverse all segments (cycles) relative to C. Due to the path starting property, new potential type-2 pairs are always found at the beginning of a new simple path in the DFS traversal. In each backtracking step, it checks whether for the current vertex v there is a triple (h, a, b) with $a = v$ on top of TSTACK and whether it corresponds to a type-2 pair. To keep track of the cycle currently examined artificial end-of-stack (EOS) markers are used. This allows to easily pop triples that violate the first descendant property which is done when backtracking over every path-starting edge. So until here we can assume that all conditions except (FC_1) and (FC_2) are satisfied. We will now explain the respective operations on TSTACK that make sure that any triple (h, a, b) for which one of the two conditions is violated will be removed before the vertex a is reached.

Lemma 5. *Suppose there is a triple (h, a, b) on TSTACK such that for the potential pair $\{a, b\}$ there is a violation of (FC_1). Then the triple will be correctly removed before the vertex a is reached on the backtracking path of DFS.*

Proof. As (FC_1) is violated, there must be a frond $x \hookrightarrow y$ such that $num(r) \leq num(x) < num(b)$. Two possible cases exist: Either x is on the half-cycle p and $x \hookrightarrow y$ is a single-edge segment, or x lies in a segment S relative to p. In both

cases, the respective segment has not yet been traversed by DFS because we are backtracking on a path of first descendants of r.

In the latter case, let $b_S \to w_S, b_S \in p \cup S, w_S \in S$ be the corresponding path-starting edge (cf. Fig. 4). Then the lowest numbered vertex of S is $a_S = lowpt_1(w_S)$ and the number of the highest numbered vertex is $h_S = num(w_S) + |D(w_S)| - 1$. As $a_S < num(a)$, a violation of (FC_1) is detected. The algorithm will perform the following update on TSTACK [7]:

- remove all triples (h, a, b) for which $a_S < num(a)$ from TSTACK
- Compute $h_{max} = max\{h|(h, a, b) \text{ removed}\}$ and $h_{new} = max\{h_{max}, h_S\}$
- push (h_{new}, a_S, b) on TSTACK

Clearly, (FC_1) is not violated for the new triple and the vertex h_{new} is the one with the highest number belonging to the split component if $\{a, b\}$ is to be split.

The update operation for single-edge segments $x \hookrightarrow y$ (which always start a path) is similar: If $num(y) < num(a)$ for a top triple (h, a, b) the violated triples are removed first. Then h_{max} is computed as above and (h_{max}, w, b) is pushed on TSTACK, as $h_{max} > x$ is always true. □

If $\{a_S, b_S\}$ ($\{y, x\}$) is the first potential pair found at the currently examined cycle or if $a_S \geq num(a)$ ($y \geq num(a)$), there is no violation of (FC_1) and (h_S, a_S, b_S) ((x, y, x)) is pushed on TSTACK.

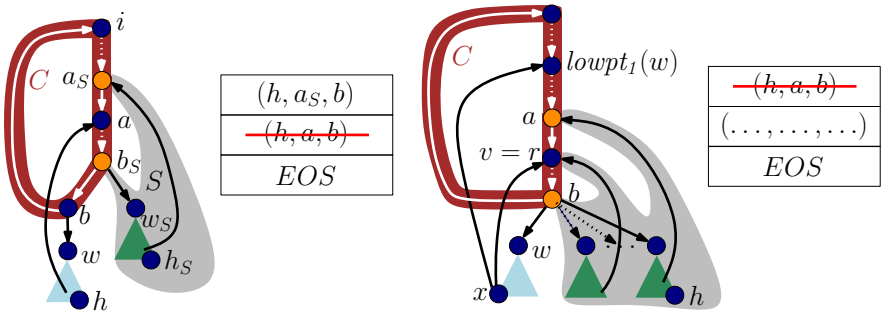


Figure 4: TSTACK operations: Update due to a violation of (FC_1) (left) found at edge $b_S \to w_S$ and triple removal due to a violation of (FC_2) found at vertex v.

For maintaining (FC_2), the algorithm makes use of the so-called highpoint $highpt(v) = max\{0, num(w)|w \hookrightarrow v\}$.

Lemma 6. *Let $a \to r \to^* v \to^* b$ be a path of first descendants. Suppose DFS is backtracking on the respective tree arc leaving v and (h, a, b) is the top triple on TSTACK. Further, suppose that $highpt(v) > h$. Let x be the start vertex of the edge $x \hookrightarrow v$ such that $highpt(v) = num(x)$. Then there exists a tree arc $b \to w$ such that $lowpt_1(w) < num(a)$ and x is a descendant of w.*

Proof. Due to the updates on TSTACK described above, we know that h is the highest numbered vertex of all segments S with vertices w that satisfy $b \to w$ and $lowpt_1(w) \geq num(a)$. Further, it is always true that $h \geq num(b)$, so if $num(x) > h$ then also $num(x) > num(b)$.

Since $num(x) > num(b)$ and b is a first descendant of r, x must be a descendant of b. Hence, x is reachable from b via tree arcs. If $b \rightarrow x$, set $w = x$, otherwise let w be the first vertex on the path $b \rightarrow^* x$. Due to the choice of h, if $num(x) > num(h)$ then it must hold that $lowpt_1(w) < num(a)$. □

While backtracking, the algorithm checks $highpt(v)$ of the current vertex v and removes triples (h, a, b) for which $highpt(v) > h$ (cf. Fig. 4). As a consequence from the above observations, the algorithm removes triples from TSTACK for which any of the algorithmic criteria is violated. Hence, if (h, a, b) remains on TSTACK until vertex a is reached, $\{a, b\}$ is a valid type-2 pair.

3.3.3 Preconditions for computing type-1 and type-2 pairs separately

The highpoint values have to be updated dynamically within the splitting procedure [6] which had not been done in the original algorithm. If fronds are removed from the adjacency list of a vertex due to split operations then the respective highpoint value has to be changed. This is essential for the correctness of the algorithm, since there may be type-1 and other type-2 splits which decrease a certain highpoint value before the highpoint test takes place at the respective vertex.

This also has the consequence that for any algorithm that computes type-2 pairs based on highpoint values all type-1 splits at descendants of a vertex v and updates of the respective data structures have to be performed before checking for type-2 pairs with v being the smaller numbered vertex.

4 Transformation of a single type-2 pair into a type-1 pair

4.1 A general transformation scheme

The fact that type-1 pairs can be computed much easier than type-2 pairs leads to the question whether it is possible to somehow transform type-2 into type-1 pairs. In this section we present such a transformation. Since we know which kinds of segments may exist around a type-2 pair without violating any conditions (recall $(S1)$ to $(S3)$ from Sect. 3.2), we construct a 'generic 'palm tree T with a type-2 pair. The central idea is to build a palm tree T' where the edges belonging to the type-2 split component in T are split as a type-1 component. We only have to make two case distinctions, namely the case where the root of T is part of the respective cycle and the case where it is not (cf. Fig. 5 and 6). Further, T has 'optional' segments and arcs to cover different scenarios. If we can transform the pair in the generic scenarios, then any type-2 pair comprising multiple or less of these segments can be transformed in the same manner.

Theorem 5. *Let T be a palm tree of a biconnected graph $G = (V, E)$ and $\{a, b\}$ a type-2 pair with $num(a) < num(b)$ in T. Then there exists another palm tree T' of G where $\{a, b\}$ is a type-1 pair and the same split components are created.*

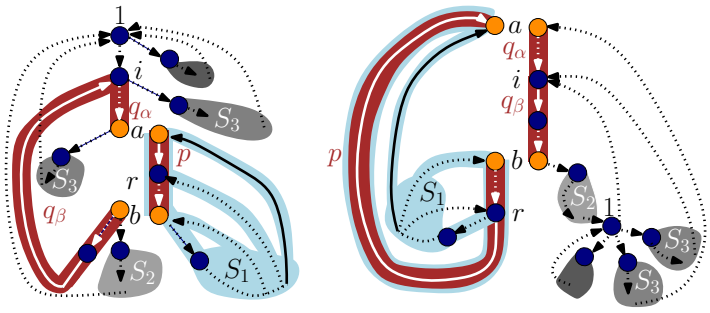


Figure 5: Transforming T with a type-2 pair (left) into T' (right).

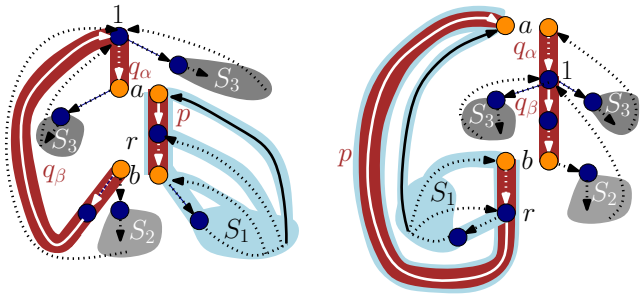


Figure 6: Transforming T with a type-2 pair and the root being part of the cycle (left) into T' (right).

Proof. For both scenarios, let a be the root of T' and q, instead of p, the tree-arc path to b. First, suppose the root is not part of the cycle and let i be the start and end vertex of the cycle in T. Then, there must be ancestors of i, i.e., a path $1 \rightarrow^* i$ which we shall call I. Due to biconnectivity, either segments of types (S_2) or (S_3) or both connect to I by fronds. In T', segments of type (S_2) reach the vertices of I via tree arcs from b and now there must be at least one frond to i like indicated in Fig. 5. Similarly, segments of type (S_3) are entered by the edges that form fronds to I in T and have fronds back to vertices on q_β.

Otherwise, if the root is part of the cycle, it will be positioned in between a and b in T' (cf. Fig. 6). Segments of types (S_2) are again reached via tree arcs from b and still have a frond to the root of T. Compared to T, segments of type (S_3) are just traversed in the inverse direction.

Finally, consider segments of type (S_1) mapped to T' for both scenarios. As p starts in b and ends with $r \hookrightarrow a$, segments of type (S_1) have their tail on p and a frond to b. Let v be the parent of b in T, so that v is now the successor of b on p in T'. From the construction of T' it follows that, independently from the exact numbering assigned to its vertices, $num(a) = 1$, $lowpt_1(v) = a$ and $lowpt_2(v) = b$. Therefore, $\{a, b\}$ is a type-1 pair in T'. Further, by the construction of p and the way segments of type (S_1) are related to p, they belong to the corresponding split component, in the same way they did in T. □

The transformation scheme can be used to eliminate single, but not multiple type-2 pairs. Further, we have no knowledge about the existence of other type-2 pairs in the target palm tree. So the question arises, whether it is possible to

construct a *type-2-free* palm tree for *every* graph. We will now give a counter example graph for which no such palm tree exists.

4.2 Not every graph has a palm tree without type-2 pairs

Consider the biconnected graph G in the left of Fig. 7. We use G as an example to prove that not every graph can be represented by a type-2-free palm tree. As can be seen directly, G has the separation pairs $\{1,5\}, \{1,6\}, \{2,5\}$ and $\{2,6\}$ and is symmetric. Due to the conditions stated in Sect. 3.2 cycles relevant for the detection of type-2 pairs must consist of at least four vertices.

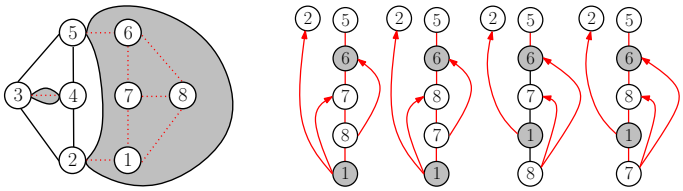


Figure 7: Example graph G with a cycle consisting of four vertices (left) and the parts of the possible palm trees corresponding to the bigger segment.

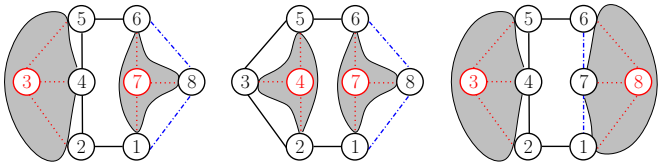


Figure 8: Situations for a cycle consisting of six vertices.

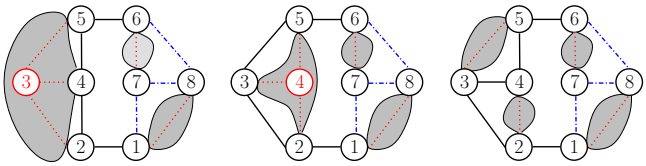


Figure 9: Cases for a cycle with seven (two leftmost) and eight (right) vertices.

In order to construct a cycle with four vertices in G, we can only use either the left or right ones. Suppose we have a palm tree for the left cycle. Independently from the order of its vertices in the tree, there will be a tree arc from this cycle to the unexplored segments on the right side. We choose $5 \to 6$ to be this tree arc and plot all possible resulting orders of the tree in the right of Fig. 7. In any of these, $\{6,1\}$ is a type-2 pair. As directing reversely, or choosing $2 \to 1$ to be the tree arc is symmetric, we are done for the four-vertices case.

For cycles consisting of six vertices we may again exploit symmetry and restrict ourselves to the cases shown in Fig. 8. In any of these, there are two unexplored vertices, say x on the left and y on the right side. They build segments, each with two fronds to vertices $\{a,b\}$ on C. The paths between

a and b on C are half-cycles (marked dash-dotted for y) and the mentioned segments are the only ones that connect to them. Hence, if we choose the root to be on the left, then $\{6, 1\}$ is a type-2 pair, otherwise $\{2, 5\}$ is a type-2 pair.

If we add another vertex to C, we obtain one segment similar to the six-vertices case and two single-edge segments (i.e. fronds) forming a chain. Looking back to Fig. 7 reminds us that these frond-chains correspond to type-2 pairs. So again the question whether $\{2, 5\}$ or $\{6, 1\}$ is to be detected as a type-2 pair only depends on the choice of the root vertex. The same holds for a cycle with eight vertices leading to frond-chains on both sides, as shown in the right of Fig. 9. We can conclude that every palm tree of G has at least one type-2 pair.

5 Conclusion

Based on the Hopcroft-Tarjan algorithm to compute the triconnected components of a biconnected multigraph, we presented a detailed analysis of separation pairs and the relationship between their graph-theoretic definition and their algorithmic detection, with emphasis on the more complex type-2 case. We also proved the correctness of certain parts of the algorithm in a less technical way using immediate implications from the structure of palm trees and the respective algorithmic detection criteria.

We investigated special cases and properties of separation pairs that were previously not covered by the common characterizations. Especially, we pointed out that the path starting property is crucial to correctly determine type-2 pairs. As long as this property is maintained, the algorithm could be modified to split type-1 and type-2 pairs independently from each other (e.g., in different passes), as long as type-1 splits precede type-2 splits.

As another contribution, we introduced a transformation scheme to turn a type-2 pair in a palm tree T into a type-1 pair in a different palm tree T'. However, we have also shown that we cannot easily exploit this scheme for simpler algorithms as there are graphs which do not admit a type-2-free palm tree. Nevertheless, we believe that the relationship inherent to this transformation could be a starting point to gain new insights which may lead to further simplifications.

Acknowledgements

I would like to thank Carsten Gutwenger for pre-reviewing parts of this work and his implementation of the Hopcroft-Tarjan algorithm in the OGDF library which was very helpful for experiments. Further, I thank Michael Jünger, Martin Gronemann, Gregor Pardella and Daniel Schmidt for a lot of fruitful discussions.

References

[1] G. Di Battista and R. Tamassia, *On-line maintenance of triconnected components with spqr-trees*, Algorithmica **15** (1996), no. 4, 302–318.

[2] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia, *Optimal upward planarity testing of single-source digraphs*, SIAM J. Comput. **27** (1998), no. 1, 132–169.

[3] R. Diestel, *Graph theory*, Graduate Texts in Mathematics, vol. 173, Springer, July 2010.

[4] D. Fussell, V. Ramachandran, and R. Thurimella, *Finding triconnected components by local replacement*, SIAM J. Comput. **22** (1993), 587–616.

[5] C. Gutwenger, *Application of spqr-trees in the planarization approach for drawing graphs*, Ph.D. thesis, Technical University of Dortmund, Germany, 2010.

[6] C. Gutwenger and P. Mutzel, *A linear time implementation of spqr-trees*, Graph Drawing (London, UK) (J. Marks, ed.), LNCS, vol. 1984, Springer, 2000, pp. 77–90.

[7] J. E. Hopcroft and R. E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput. **2** (1973), no. 3, 135–158.

[8] G. Kant, *Drawing planar graphs using the canonical ordering*, Algorithmica **16** (1996), no. 1, 4–32.

[9] M. Mader, *Planar graph drawing*, Master's thesis, Universität Konstanz, Germany, 2008.

[10] G. L. Miller and V. Ramachandran, *A new graph triconnectivity algorithm and its parallelization*, Combinatorica **12** (1992), no. 1, 53–76.

[11] A. M. Saifullah and A. Üngör, *A simple algorithm for triconnectivity of a multigraph*, CATS '09: Proc. of the 15th Australasian Symposium on Computing: The Australasian Theory (Darlinghurst, Australia) (R. Downey and P. Manyem, eds.), vol. 94, Australian Computer Society, Inc., 2009, pp. 53–62.

[12] W. T. Tutte, *Connectivity in graphs*, University of Toronto Press, 1966.

[13] K.-P. Vo, *Finding triconnected components of graphs*, Linear and Multilinear Algebra **13** (1983), no. 2, 143–165.

[14] K.-P. Vo, *Segment graphs, depth-first cycle bases, 3-connectivity, and planarity of graphs*, Linear and Multilinear Algebra **13** (1983), no. 2, 119–141.

[15] S. G. Williamson, *Combinatorics for computer science*, Computer Science Press, Inc., New York, 1985.